组成原理实验课程矩阵乘法优化鲲鹏版实验报告

实验名称	矩阵乘法优化			班级	李涛
学生姓名	艾明旭	学号	2111033	指导老师	董前琨
实验地点	A306		实验时间	2023.6.3	

1、 实验目的

参考课程中讲解的矩阵乘法优化机制和原理，在 Taishan 服务器上使用 vim+gcc 编程环境，完成不同层次的矩阵乘法优化作业。

2、 实验内容说明

• 使用鲲鹏服务器，使用基本矩阵乘法、指令集并行矩阵乘法、分块矩阵乘法、多处理器矩阵乘法，在 1024、2048、3072、4096 数量级下比较他们执行速度

• 总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行性能、加速比等（概述本次实验要做什么，参见实验要求）

3、 加速原理

3.1 指令级并行

通过同时执行多条指令来提高计算性能的技术。在矩阵乘法中，可以使用 SIMD（单指令多数据）指令集，如 AVX，同时处理多个数据元素。通过将矩阵元素打包成向量形式，并使用适当的指令集，可以在**一次指令执行中完成多个**乘法和累加操作。如流水线、超标量、超长指令字等。

3.2 分块处理

当矩阵尺寸过大时，由于 Cache 的局部性原理，数据访问会倾向于访问附近的数据，但数据的大小大大超过 Cache，使得**命中率过低**，导致处理器需要多次访问内存造成时间浪费。通过将矩阵分割为适当大小的块然后**按块计算**，可以减少数据访问的跨越，在一个子矩阵（块）被 cache 1 替换出去之前，最大限度的对其进行数据访问。利用时间局部性与缓存的局部性原理来提高性能，提高 cache 命中率。

3.3 多处理器并行的分块

调度多核处理器，将矩阵分割为多个块，并**分配**给不同计算节点，每个处理器或计算节点可以独立地计算其分配的块，最后合并结果。这样可以充分利用并行计算资源，加快整个矩阵乘法的计算速度。

4、 实验步骤

由于鲲鹏的 AVX 库进行子字优化在 Taishan 服务器上的软件包环境不好配置，这里不进行子字并行优化，删除原 128 位的数，__m256d 换位 double 类型，删除子字并行函数 avx_dgemm （）及测试数据。代码如下：

```
#include<iostream>
#include<time.h>
//#include<x86intrin.h>
#include<immintrin.h>
```

```
    using namespace std;
    #define REAL_T double

    void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t
stop ){
        REAL_T flops = ( 2.0 * A_height * B_width * B_height ) / 1E9 /((stop -
start)/(CLOCKS_PER_SEC * 1.0));
        cout<<"GFLOPS:\t"<<flops<<endl;
    }

    void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C ){
        for( int i = 0; i < n; ++i )
            for( int j = 0; j < n; ++j ){
                A[i+j*n] = (i+j + (i*j)%100 ) %100;
                B[i+j*n] = ((i-j)*(i-j) + (i*j)%200 ) %100;
                C[i+j*n] = 0;
            }
    }

    void dgemm( int n, REAL_T *A, REAL_T *B, REAL_T *C){
        for( int i = 0; i < n; ++i )
            for( int j = 0; j < n; ++j ){
                REAL_T cij = C[i+j*n];
                for( int k = 0; k < n; k++ ){
                    cij += A[i+k*n] * B[k+j*n];
                }
                C[i+j*n] = cij;
            }
    }

    void avx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
        for( int i = 0; i < n; i+=4 )
            for( int j = 0; j < n; ++j ){
                __m256d cij = _mm256_load_pd( C+i+j*n );
                for( int k = 0; k < n; k++ ){
                    //cij += A[i+k*n] * B[k+j*n];
                    cij = _mm256_add_pd(
                            cij,
                            _mm256_mul_pd( _mm256_load_pd(A+i+k*n),
_mm256_load_pd(B+i+k*n) )
                            );
                }
                _mm256_store_pd(C+i+j*n,cij);
            }
```

```
    }

    #define UNROLL (4)

    void pavx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
        for( int i = 0; i < n; i+=4*UNROLL )
            for( int j = 0; j < n; ++j ){
                __m256d cij[4];
                for( int x = 0; x < UNROLL; ++x)
                    cij[x]= _mm256_load_pd( C+i+j*n );

                for( int k = 0; k < n; k++ ){
                    //cij += A[i+k*n] * B[k+j*n];
                    /*cij = _mm256_add_pd(
                            cij,
                            _mm256_mul_pd( _mm256_load_pd(A+i+k*n),
_mm256_load_pd(B+i+k*n) )
                            );*/
                    __m256d b = _mm256_broadcast_sd( B+k+j*n );
                    for( int x = 0; x <UNROLL; ++x)
                        cij[x] = _mm256_add_pd(
                            cij[x],
                            _mm256_mul_pd( _mm256_load_pd(A+i+4*x+k*n),  b ) );
                }
                for( int x = 0; x < UNROLL; ++x)
                    _mm256_store_pd( C+i+x*4 +j*n, cij[x]);
            }
    }

    #define BLOCKSIZE (32)
    void do_block( int n, int si, int sj, int sk, REAL_T *A, REAL_T *B, REAL_T *C){
        for( int i = si; i < si + BLOCKSIZE; i+=UNROLL*4 )
            for( int j = sj; j < sj + BLOCKSIZE; ++j){
                __m256d c[4];
                for( int x = 0; x < UNROLL; ++x )
                    c[x] = _mm256_load_pd( C+i+4*x+j*n );

                for( int k = sk; k < sk + BLOCKSIZE; ++k ){
                    __m256d b = b = _mm256_broadcast_sd( B+k+j*n );
                    for( int x = 0; x <UNROLL; ++x)
                        c[x] = _mm256_add_pd(
                            c[x],
                            _mm256_mul_pd( _mm256_load_pd(A+i+4*x+k*n),  b ) );
                }
```

```
                for( int x = 0; x < UNROLL; ++x)
                    _mm256_store_pd( C+i+x*4+j*n, c[x]);
            }
    }


    void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
        for( int sj = 0; sj <n; sj+=BLOCKSIZE)
            for( int si = 0; si <n; si+=BLOCKSIZE)
                for( int sk = 0; sk <n; sk+=BLOCKSIZE)
                    do_block( n, si, sj, sk, A, B, C);
    }

    void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
    #pragma omp parallel for
        for( int sj = 0; sj <n; sj+=BLOCKSIZE)
            for( int si = 0; si <n; si+=BLOCKSIZE)
                for( int sk = 0; sk <n; sk+=BLOCKSIZE)
                    do_block( n, si, sj, sk, A, B, C);
    }

    void main()
    {
        REAL_T *A, *B, *C;
        clock_t start,stop;
        int n = 1024;
        A = new REAL_T[n*n];
        B = new REAL_T[n*n];
        C = new REAL_T[n*n];
        initMatrix(n, A, B, C);

        cout<< "origin caculation begin...\n";
        start = clock();
        dgemm( n, A, B, C );
        stop = clock();
        cout <<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C);
        cout<< "AVX caculation begin...\n";
        start = clock();
        avx_dgemm( n, A, B, C );
```

```
        stop = clock();
        cout <<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C);
        cout<< "parallel AVX caculation begin...\n";
        start = clock();
        pavx_dgemm( n, A, B, C );
        stop = clock();
        cout <<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C);
        cout<< "blocked AVX caculation begin...\n";
        start = clock();
        block_gemm( n, A, B, C );
        stop = clock();
        cout <<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C);
        cout<< "OpenMP blocked AVX caculation begin...\n";
        start = clock();
        omp_gemm( n, A, B, C );
        stop = clock();
        cout <<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";
        printFlops(n, n, n, start, stop);
    }
```

5、实验结果分析

5.1 1024

```
stu2111252@parallel542-taishan200-1:~$ vim l
stu2111252@parallel542-taishan200-1:~$ g++ l
stu2111252@parallel542-taishan200-1:~$ ls
a.out  ljh.cpp  mat.cpp
stu2111252@parallel542-taishan200-1:~$ a.out
a.out: command not found
stu2111252@parallel542-taishan200-1:~$ ./a.o
origin caculation begin...
18.69487                GFLOPS: 0.118846
parallel AVX caculation begin...
14.421329               GFLOPS: 0.14891
blocked AVX caculation begin...
7.434           GFLOPS: 0.306764
OpenMP blocked AVX caculation begin...
7.8429          GFLOPS: 0.306414
stu2111252@parallel542-taishan200-1:~$
```

图 1: 1024

5.2 2048

```
stu2111252@parallel542-taishan200-1:~$ vim
stu2111252@parallel542-taishan200-1:~$ g++
stu2111252@parallel542-taishan200-1:~$ ./a.
origin caculation begin...
217.411218              GFLOPS: 0.0790202
parallel AVX caculation begin...
166.113829              GFLOPS: 0.103422
blocked AVX caculation begin...
56.520805               GFLOPS: 0.303957
OpenMP blocked AVX caculation begin...
56.348548               GFLOPS: 0.304886
```

图 2: 2048

5.3 3072

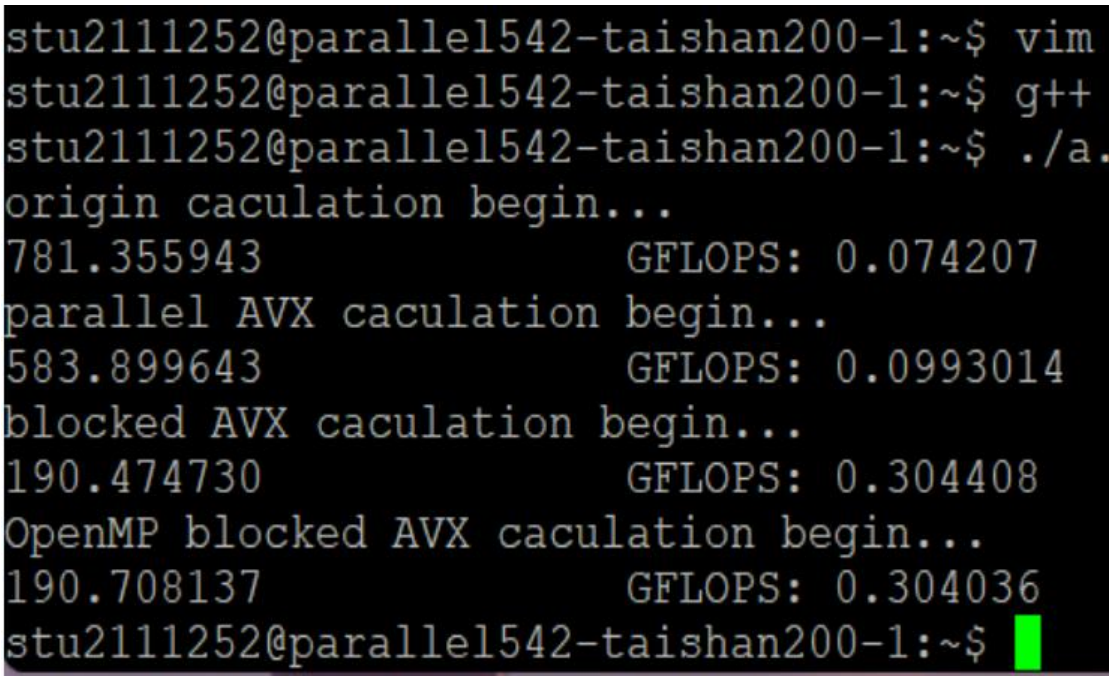

图 3: 3072

5.4 4096

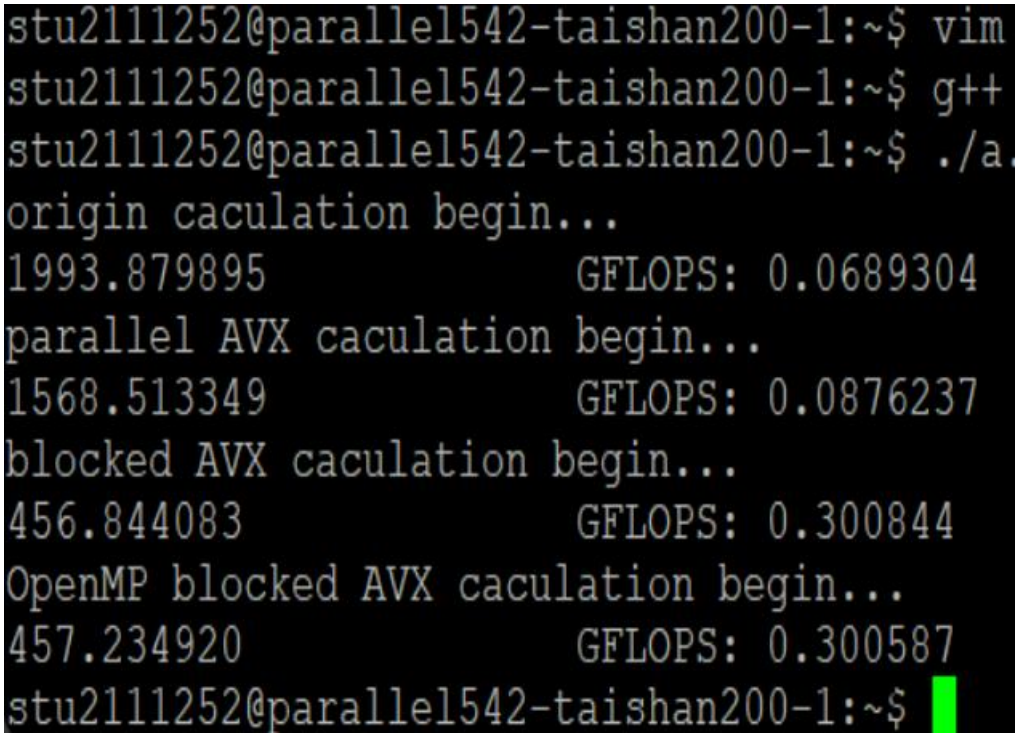

图 4: 4096

6 对比分析

1. 从实验数据上看，泰山服务器的运算速度远不及我们本地计算机，这里从两个硬件配置进行比较。以原 Orgin 和 Block 在 size 为 2048 和 4096 下为例比较

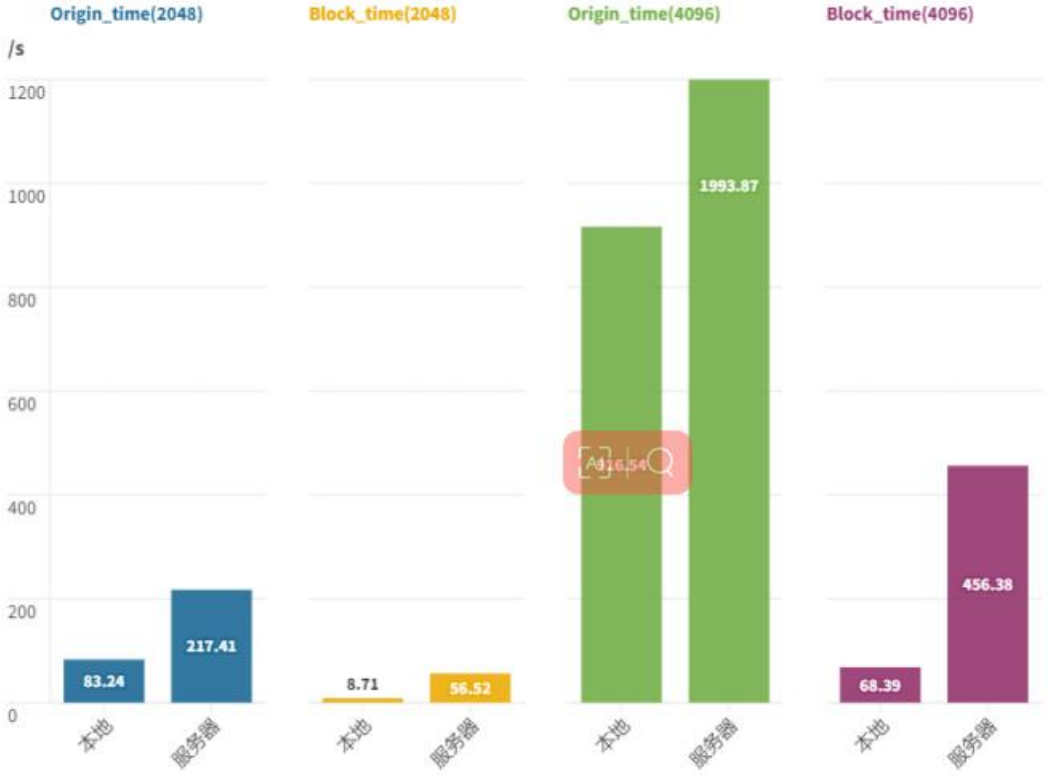

图 5: 本地和鲲鹏的 Time 比较

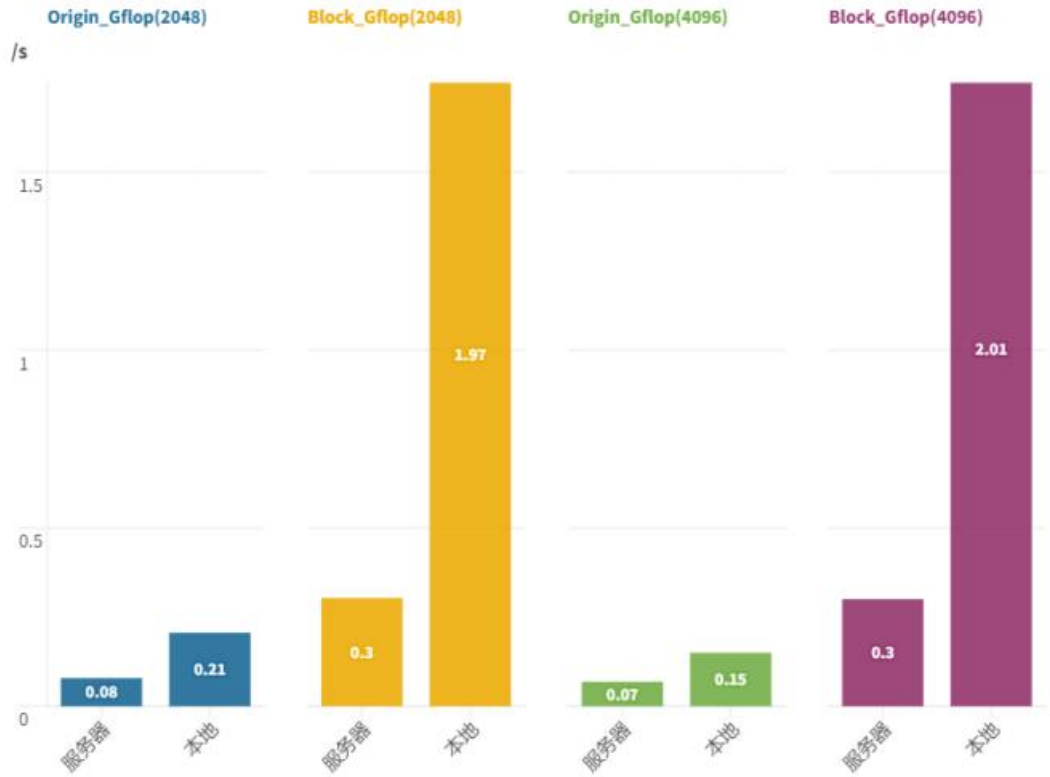

图 6: 本地和鲲鹏的 GFlop 比较

• 我的电脑处理器为 11th Gen Intel(R) Core(TM) i7-11800H，时钟频率达到 2.3GHz，机带 RAM16G，而华为鲲鹏 920 处理器，支持 2 路处理器，处理器包含 64 核，48 核，40 核和 32 核三种配置，频率均为 2.6GHz。我的电脑 13cache 是 24MB，而服务器 64MB。
• 处理器上看起来泰山要比我的电脑好（不管是 13 缓存还是处理器性能），但或许它是针对服务器设计的，具体为什么它比我的本机慢，或许还跟我使用的 vs 编译器对我的代码进行优化有关；此外据我了解，intel 在单核处理上的能力是要大于 x86 的。

2. 其运行时间、加速度、每秒浮点运算次数比较如下

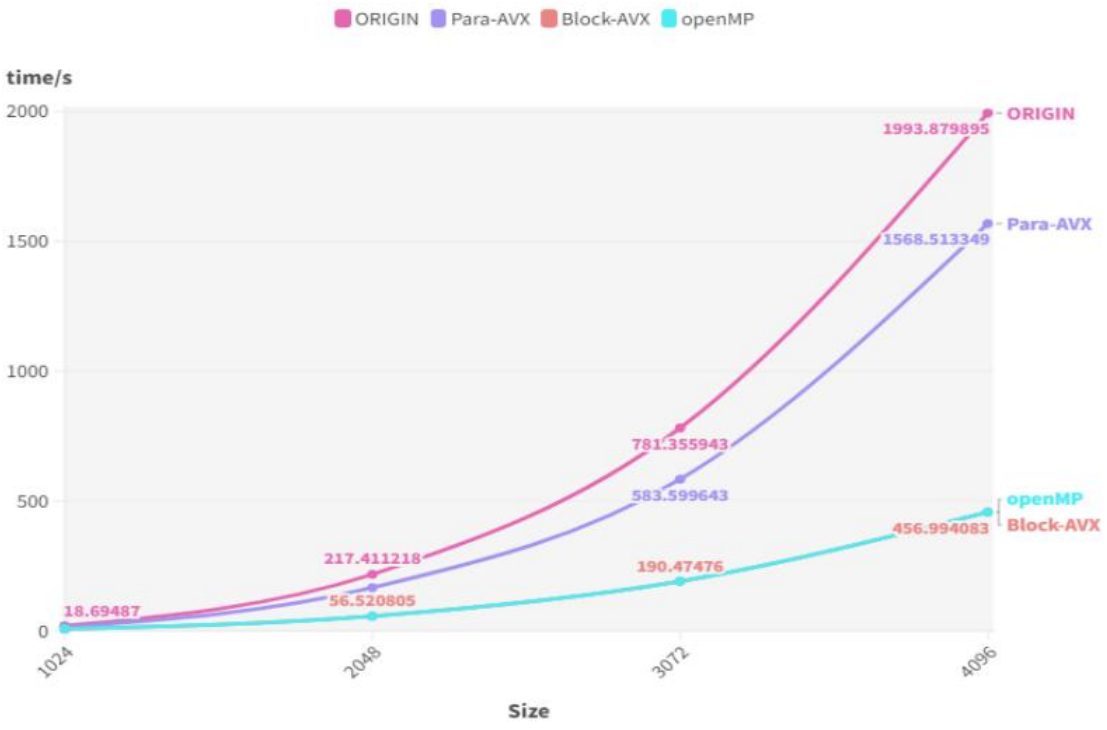

图 7：运行时间

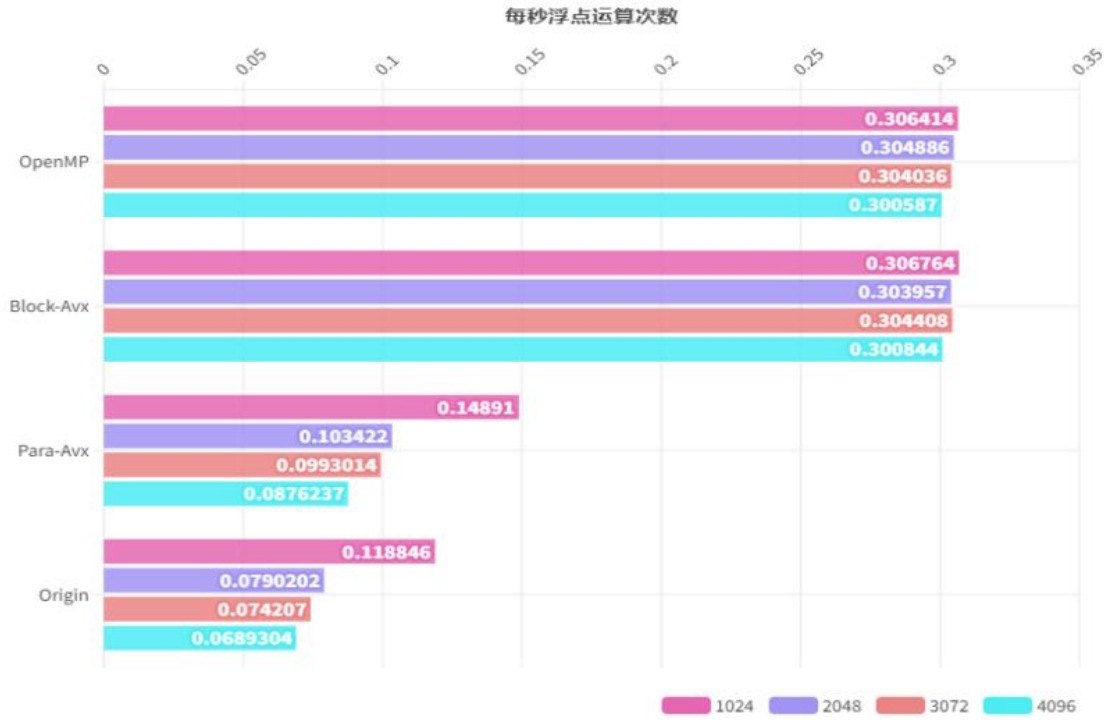

图 8：每秒浮点运算数

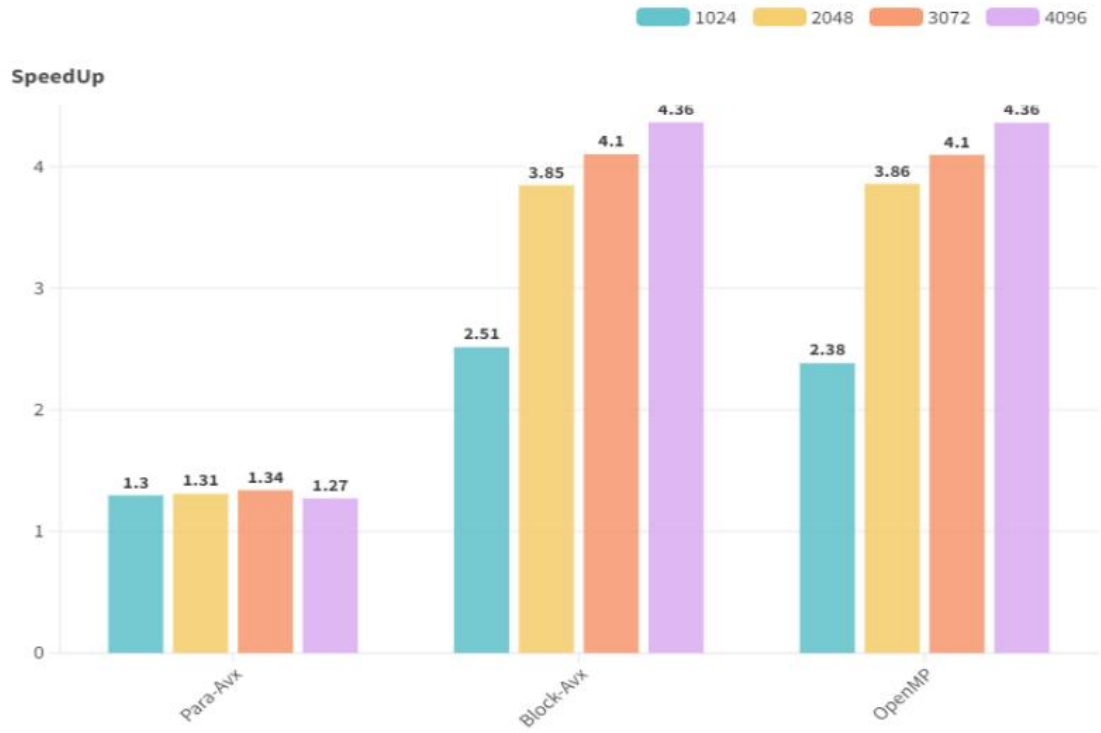

图 9：SpeedUp

首先从运行时间上，这四种方法的时间增量都比较大（相比我本地的那个曲线而言），origin 是 n3 次方的曲线，而分块并没有显著降低其时间。可能是服务器的 cache 给我们开放的本身就较小或者其他原因。

• 其次，最后使用 openmp，即多处理器并行分块和单处理器的并行分块时间相同，这里我使用的编译指令 g++ -fopenmp ljh.cpp 是成功的，但是效果出来却无效，而看鲲鹏配置是有**支持多处理器并行的**，原因应该是服务器只给我分配了**单核处理器**。

• 其次，观察时间 orign 的算法复杂度是 n 立方，但从时间上看，其时间增量大于 n 的立方，原因可能位因为矩阵 size 增大后，其向下一级缓存访问的次数增大，所以时间会有额外花费。

• 最后在矩阵较小时，发现其加速比增量不明显，原因是其 io 花费时间对总时间的影响更大，因此加速比增加不明显。

7.总结感想

在我原本的思路里，我认为同样的程序让服务器来执行，其性能远远超出我们自己的电脑。但是实际情况还需要对比一下程序执行的硬件环境和软件环境。比如这次矩阵乘实验，服务器只给我们单核且分配的 Cache 也不足，而我们本地是支持多核且有软件的代码优化。

总之矩阵乘优化实验给我带来最大收获就是从硬件对于性能提升十分重要，自己写代码时要考虑如何更少造成数据通路堵塞、如何更大程度利用 cache 的局部性原理，并且大规模计算使用并行设计、GPU 加速十分重要！

感谢理论课老师李涛老师与实验课指导老师董前琨老师，还有每一位助教学长学姐，感谢这一学期的认真指导与悉心传授，我会更加进一步对计算机相关原理知识探究，并将其用在我今后的学习与工作中。